Polymer: A Model-Driven Approach for Simpler, Safer, and Evolutive Multi-Objective Optimization Development

Assaad Moawad[1], Thomas Hartmann[1], François Fouquet[1], Grégory Nain[1], Jacques Klein[1], and Johann Bourcier[2]

[1]*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg*
firstname.lastname@uni.lu

[2]*IRISA / INRIA, University of Rennes 1, France*
johann.bourcier@inria.fr

Keywords: Multi-Objective Evolutionary Algorithms, Optimization, Genetic Algorithms, Model-Driven Engineering.

Abstract: Multi-Objective Evolutionary Algorithms (MOEAs) have been successfully used to optimize various domains such as finance, science, engineering, logistics and software engineering. Nevertheless, MOEAs are still very complex to apply and require detailed knowledge about problem encoding and mutation operators to obtain an effective implementation. Software engineering paradigms such as domain-driven design aim to tackle this complexity by allowing domain experts to focus on domain logic over technical details. Similarly, in order to handle MOEA complexity, we propose an approach, using model-driven software engineering (MDE) techniques, to define fitness functions and mutation operators without MOEA encoding knowledge. Integrated into an open source modelling framework, our approach can significantly simplify development and maintenance of multi-objective optimizations. By leveraging modeling methods, our approach allows reusable optimizations and seamlessly connects MOEA and MDE paradigms. We evaluate our approach on a cloud case study and show its suitability in terms of *i)* complexity to implement an MOO problem, *ii)* complexity to adapt (maintain) this implementation caused by changes in the domain model and/or optimization goals, and *iii)* show that the efficiency and effectiveness of our approach remains comparable to ad-hoc implementations.

1 INTRODUCTION

In many domains, such as finance, science, engineering, and logistics several conflicting objectives need to be simultaneously optimized. In finance for example, the anticipated value of a portfolio should be maximized, whereas the expected risk should be minimized. Such problems, involving conflicting objectives, are called *multi-objective optimization (MOO) problems*, characterized by potential solutions offering trade-offs between different objectives. Different algorithms can cope with such problems, *e.g.* particular swarm optimization (Kennedy et al., 1995), simulated annealing (Van Laarhoven and Aarts, 1987), and population based algorithms (Deb et al., 2002). Multi-objective evolutionary algorithms (MOEAs) are another class of algorithms, which has proved to be particularly capable of finding solutions for complex domain-specific optimization problems with large solution spaces for which typically no efficient deterministic algorithms exist. However, MOEAs are difficult to use, require specific and detailed expert knowledge about fitness functions, mutation operators, and genetic problem encodings. Common encodings consist in mapping a domain-specific MOO problem into a binary, permutation-based matrix, or graph-based representation (Coello et al., 2002). The solutions found by MOEAs must then be decoded, meaning to map them back to the domain-specific multi-objective optimization problem. This makes it very challenging to properly apply MOEAs and may require developers to focus more on the encoding of a problem than on the problem itself (Konak et al., 2006). The continuous design process of today's software systems makes it even more difficult to implement and especially to maintain MOEAs. Each change in the domain and/or in the optimization goals requires to adapt the problem encoding and decoding. This makes it first necessary to identify which impacts a change has on the problem encoding. Since the encoding is usually not statically typed, type checkers cannot be used to indicate potential errors.

As a consequence it is hardly possible to use standard refactoring techniques to apply domain and/or optimization goal changes. Instead, the problem encoding must be adapted manually and independently from the domain representation. Moreover, optimization problems are inevitably linked to the growing complexity of software.

In this paper we present a new MDE approach to develop MOO layers directly on top of domain models. Fitness functions and mutation operators can use these models and their API instead of relying on complex and error prone encoding steps. Similar to paradigms like domain-driven design, our model-driven approach allows developers to focus on the actual domain-specific optimization problems rather than on technical encoding details. Our approach also reduces the gap between MOEA representations and models, allowing to reuse standard modeling tools (*e.g.* model checkers) within fitness functions or mutation operators. Integrated into an open-source modeling framework, our approach can significantly simplify domain-specific MOO development. We evaluate our approach on a cloud case study and show its suitability in terms of *i)* complexity to implement a MOO problem, *ii)* complexity to adapt (maintain) this implementation caused by changes in the domain model and/or optimization goals, and *iii)* efficiency and effectiveness of our approach remains comparable to traditional implementations.

This paper is organized as follows. Section 2 introduces MOEA background. In section 3 we provide a case study, which we further use to present our contribution in section 4 and to evaluate it in section 5. The related work is discussed in section 6. Finally, section 7 gives the conclusion and future work.

2 BACKGROUND

In the 1960s, several researchers independently suggested to adopt the principles of natural evolution (Darwin's theory) for optimizations. This created the field of evolutionary algorithm (EA). In EA, a solution vector is called *individual* or *chromosome*, which consist of discrete units, which are called *genes*. Each gene controls one or more features of a solution. Usually, a chromosome corresponds to a unique solution in the solution space. This requires a mapping, called *encoding*, between the domain specific solution space and chromosomes. Usually, the encoding step is complex and can be even more complicated than the actual optimization problem itself (Konak et al., 2006). EAs operate on a collection of chromosomes, called a *population*. The population is usually randomly initialized and then evaluated with a provided *fitness function* in order to select the *"most appropriate"* one for the next *generation*. After this step, EAs use two operators to generate new solutions: crossover and mutation. The *crossover* operator takes two chromosomes, combines them together and finally creates a new offspring. The *mutation* operator injects random changes into chromosomes.

The first MOEA was proposed by Shaffer (Schaffer, 1985). MOEAs, unlike EAs, solve problems involving multiple conflicting objectives and offer a representative subset of the *Pareto optimal solution set*, rather than a single optimized solution. *"The Pareto optimal set is a set of solutions that are non-dominated with respect to each other. While moving from one Pareto solution to another, there is always a certain amount of sacrifice in one objective(s) to achieve a certain amount of gain in the other(s)"* (Konak et al., 2006). There are several algorithms to select which solution subsets of the Pareto optimal set to keep in order to cover, as diversely as possible, the different trade-offs between objectives. The NSGA (Deb et al., 2002) algorithm family (*e.g.* NSGA-II, NSGA-III) is amongst the most well known ones. MOEAs, like conventional EAs, rely on a genetic encoding step. However, besides mutation and crossover operators, for MOEAs several fitness functions representing the different objectives, need to be defined, instead of only one fitness function in EAs. Subsequently, a decision making procedure has to be called to select one solution from the Pareto subset.

3 REAL-WORLD CASE STUDY

MOEA frameworks are often evaluated against a suite of mathematical functions with well known characteristics like ZDT, DTLZ and LZ09 (Auger et al., 2012). These problems are well suited to evaluate the performance of MOEAs in terms of finding the best Pareto front and the optimum solution. However, they are less suitable to represent the complexity of real-world problems and to evaluate the required effort for developers to implement and maintain MOOs for domain-specific problems. In this section we define a non trivial real-world MOO case study to apply our approach to. As shown in (Frey et al.,) the cloud computing domain is an appropriate real-world case study where MOEAs are used for scheduling and scaling tasks. Scheduling applications on the cloud is a non trivial task (Pandey et al., 2010). First of all, several different cloud computing providers exist on the market. Each of these offers different computing specifications (*e.g.* CPU, memory, storage, and networking capacity) at different prices and pricing models (fixed pricing, bidding, etc). Then, different

applications can be deployed on the cloud, each having different requirements in terms of hardware (CPU, RAM, disk, network), security (such as sand-boxing, redundancy), priority, and latency (distribution on different geographic zones). All of these requirements can be translated into optimization fitness functions.

For this paper, we define the following case study: we have a fixed number $n = 7$ of applications, each requiring a specific computational power (in virtual CPU hours). Further, we assume that these applications can be parallelized and distributed into smaller tasks. Then we can rent a variable number $m < 100$ of cloud instances from a provided list of available instance models $M = 47$ (we provide as input for each: number of virtual CPUs and fixed price per hour). An instance model can be rented several times. We divide each application into m tasks and we assign a weight to each task (between 0 and 100). The weight w_{ij} of application i on instance j represents the proportion of resources allocated for the application i on the instance j. The optimization objectives are, *i)* reducing the average time required to execute all applications and *ii)* reducing the total price paid on instances. These two objectives are conflicting: the more we pay, the more instances we can rent and the less average time is needed for the execution. In addition, we define three mutation operators for this case study: *AddInstance* (to allocate a new cloud instance), *RemoveInstance* (to delete a cloud instance), and *ChangeWeight* (to randomly change the weight of an application on an instance). We implement the same problem with the same input files with our approach, using models and in a traditional way, using an array encoding with the jMetal framework (Durillo and Nebro, 2011). jMetal is an object-oriented Java-based framework for MOO with metaheuristics. We use for both implementations a $population\_size = 20$ and $max\_generations = 1000$.

4 MODEL-BASED MOO

In this section we describe our model-driven approach to simplify complex multi-objective optimizations.

4.1 Approach

The classical process of using MOEAs in an application can be divided into six steps, which are shown in figure 1. *First*, the problem domain must be defined and implemented. This is usually achieved using tools such as UML or E/R diagrams and standard programming languages like Java or C/C++ (POJO based). *The second step* is the encoding step. This means that the section of the domain impacted by the

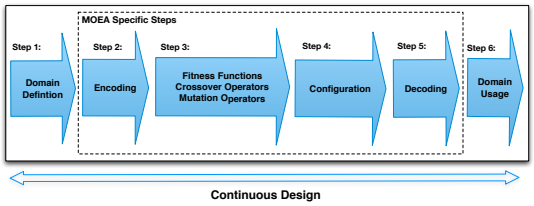

Figure 1: Classical steps to use MOEAs

MOO must be mapped to a suitable structure in order to execute MOEAs. Typically, representations like binary or int arrays, permutations, matrix, or graphs are used for this purpose (Coello et al., 2002). Figure 2 shows an example of such a classical encoding (based on an int array), aligned to our cloud case study. As

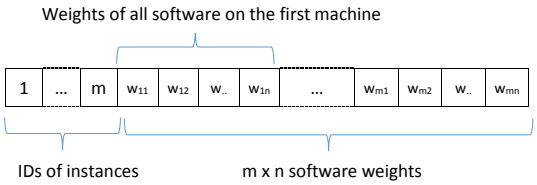

Figure 2: Classical MOO encoding using arrays

for our case study, the number of reserved instances m can vary and is subject to optimization, the size of the array varies as well. For each computing instance, we need to allocate an integer for the *ID* of the instance (in our case a number from 1 to 47), and n integers for the weights of the n applications. In this way the encoding consists of an integer array of size: $m \times (n+1)$. As a design choice, we decide that the first m values in the encoding represent the *IDs* of the reserved instances and the next $m \times n$ values represent the weights of the applications. However, this classical mapping comes with a number of drawbacks. First, this task is usually not trivial and requires expertise form both sides, domain experts as well as MOEA experts. Second, type-safety is lost. Consequently, it is more difficult to find bugs in the encoded MOEA representation and to maintain it. Last but not least, such encodings are difficult to read. *In a third step* fitness functions, crossover, and mutation operators must be defined. Listing 3 shows as an example, again aligned to our cloud case study, the implementation of a mutator to remove an instance.

Listing 1: RemoveInstance mutator on array encoding

```
class RemoveInstance implements Mutation {
  void mutate(int[] cloud) {
  int m= cloud.length/(N_SOFT+1);//number of instances
  if(m==0) return;
  int x= rand.nextInt(m);
  int[] newCloud= new int[(m-1)*(N_SOFT+1)];
  for(int i=0;i<x;i++) { newCloud[i]=cloud[i] };
  for(int i=x+1;i<m+x*N_SOFT;i++){newCloud[i-1]=cloud[i]};
  for(int i=m+(x+1)*N_SOFT;i<m*(N_SOFT+1);i++){
    newCloud[i-1-N_SOFT]=cloud[i]}; cloud=newCloud; }}
```

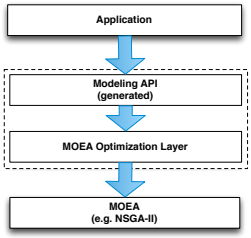

Figure 3: Model-based MOO overview

As can be seen in the listing, in order to remove a machine *x* from the cloud, we need to remove its *ID* from the *ID* section then remove the *n* associated application *weights* from the *weights* section. *Next*, the MOEA setup must be configured. This includes the initialization of the population, its size, and the selection of an algorithm for the diversification of solutions (*e.g.* NSGA-II). *In a fifth step* the solution found by a MOEA must be interpreted and decoded back to the domain representation. *Finally*, the decoded solutions can be used in the domain-specific MOO problem.

A major challenge that comes along with this classical process is that in case of refactoring, and continuous design, all these steps have to be checked for impacts and potentially must be repeated.

We claim that this process can be significantly simplified by leveraging model-driven engineering techniques to allow a domain-specific model encoding for MOO problems. The main idea of this approach is to allow to express the MOO problem, fitness functions, crossover and mutations operators directly and seamlessly using the domain model, making explicit encoding and decoding steps unnecessary. This allows to throughout use the same models for both the domain representation and MOO problem encoding and therefore to avoid this mismatch. By integrating our approach into the open-source *Kevoree Modeling Framework (KMF)* (Fouquet et al., 2012) we provide a framework[1] to express domain-specific model encodings for MOEAs in order to discard the encoding/decoding steps. Figure 3 shows an overview of the layers involved in our approach. In MDE the application layer is typically built using a modeling API, which is generated form a meta-model definition (*e.g.* from an Ecore model). In our approach, we use MOF based modeling, using textual and graphical representations. We extend this modeling API with several interfaces to implement fitness functions, mutation and crossover operators, as well as the population creation factory. These interfaces are typed with the model, which means that they can be implemented using the modeling API. Listing 2 shows a simplified version of our interfaces.

[1] http://kevoree.org/polymer/

Listing 2: Interfaces of the extended modeling API

```
interface FitnessFunction<A extends KContainer> {
  double evaluate(A model);}
interface  MutationOperator<A extends KContainer> {
    List<MutationVars> enumerateVars(A model);
    void mutate(A model);}
interface CrossoverOperator<A extends KContainer> {
    A execute(A modelA, A modelB);}
```

Figure 4 represents the model encoding of the MOO problem of our cloud case study. As can be

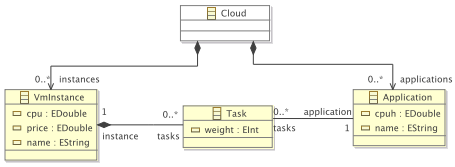

Figure 4: A cloud meta-model

seen in the figure, a cloud contains several virtual machine instances and applications, which can be executed on the virtual machine instances. Using standard model-driven techniques this meta-model can be transformed into object-oriented code. Listing 3 again shows the implementation of the *RemoveInstance* mutator. But this time the operator is implemented directly on the model encoding instead of an explicit MOO problem encoding.

Listing 3: RemoveInstance mutator on a model encoding

```
class RemoveInstance implements MutationOperator<Cloud> {
  void mutate(Cloud could) {
    int m=could.getInstances().size();
    if(m==0) return;
    int x= rand.nextInt(m);
    VmInstance vmtoRemove = could.getInstances.get(x);
    could.removeInstances(vmtoRemove); }}
```

The extended modeling API allows developers to encode the MOO problem completely in terms of the domain model. In addition, we provide a MOEA optimization layer, which initiates the population, calculates the fitnesses, and executes mutation and crossover operators according to their implementations. We use established algorithms such as NSGA-II and ε-MOEA for the actual MOO. These algorithms only operate on the provided fitnesses and are independent from our model encoding. Our optimization layer provides the fitnesses and executes the mutations (both on top of models) for the underlaying MOEA layer. The MOEA optimization layer can be configured to choose one of the supported MOEAs and to set the maximum number of generations.

This approach enables the execution of MOEAs on top of models and can significantly simplify the usage of MOEAs to tackle domain-specific MOO problems. The necessary MOEA expertise is hidden in the framework and allows software engineers to focus on

the domain-specific MOO problem, instead of MOEA encoding and decoding. First, software engineers can express the optimization problems in terms of the domain model without being MOEA experts. Second, type-safety is maintained, which improves the maintainability of the application. Third, since the problem is expressed in domain terms the solution is far more readable. Leveraging type safety and object-oriented design, MOEAs can shift to complex optimization using several mutation operators. Last but not least, our approach enables to use standard modeling techniques during the optimization process, and thus can leverage many tools to verify solutions.

4.2 Polymer Implementation

We implement our framework on top of KMF (Fouquet et al., 2012), which is an alternative to the Eclipse Modeling Framework (EMF), fully supporting the Ecore file format but targeting runtime performance. We use KMF to generate the modeling code and API from domain models to support the construction of MOEA elements (fitness functions and mutation operators) on top. The Polymer framework core itself on one hand provides the MOEA interfaces described in section 4.1 and on the other hand implements a mapping layer to intermediate between classical MOEAs and our model-based encoding. Therefore, Polymer calculates the fitness function score based on the implementation of the provided interfaces, using a classical visitor pattern. The result of the different fitness functions (plain numbers) are used as input for classical MOEA Pareto-front selectors. Whenever a MOEA needs to mutate one of the solutions, Polymer creates a clone of the domain model and executes one of the implemented mutation operators (randomly) on this clone. Since cloning of models is a frequent but costly operation in our approach we implemented an efficient partial cloning mechanism, which is described in more detail in section 4.3. Like fitness functions, mutation operators operate directly on the generate modeling API and can take advantage of model transformations. In this paper we do not intent to provide a new MOEA algorithm, instead we allow to use different already existing search algorithms. The novelty of our approach is to allow the usage of a model-based encoding for domain specific MOO problems. Figure 5 summerizes the implementation concept of Polymer.

4.3 Partial Model Cloning

It is easy to efficiently copy traditional MOEA problem encodings, *e.g.* binary arrays. However, our

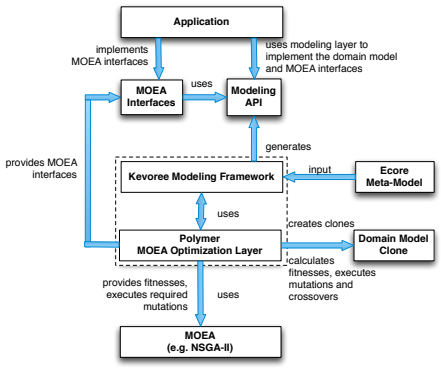

Figure 5: Model-based MOO implementation

model-based problem encodings are more difficult and less efficient to copy, since potentially a complete model must be cloned. To reach a comparable efficiency in terms of memory usage and performance, while still offering developers the advantages of working directly on top of domain models, we provide a partial model cloning mechanism (Fouquet et al., 2014). First, for most real-world applications, we argue that only a fraction of parameters and fields of domain-specific models are subject to optimization. A big part of the model consists of immutable fields or reflects static characteristics of the domain and consequently is outside the search space of MOEAs. Therefore, whenever we clone a model to generate another potential solution, we ensure in our framework to not copy immutable fields but only refer to them. In fact, this mechanism only clones the mutable parts of a model (partial cloning) instead of the whole model. For our case study, the application class can be declared as immutable, because the mutations do not impact this class. In the cloud case study, the optimization occurs on the rented machine and the way the weights are distributed. This partial cloning is completely transparent for developers.

5 EVALUATION

We evaluate our approach on the cloud case study and show its suitability in terms of *i)* complexity to implement a MOO problem, *ii)* complexity to adapt (maintain) this implementation, and *iii)* show that the efficiency and effectiveness of our approach remains comparable to ad-hoc implementations.

5.1 Complexity to Implement

We claim that a major complexity of MOEA based developments is due to the required mixed expertise, on the one hand MOEA and on the other hand the

Feature	jMetal		Polymer	
	M. LoC	D. LoC	M. LoC	D. LoC
Domain Encoding	1	–	–	24
Fitness Operators	13	39	–	39
Mutation Operators	28	32	–	34
Crossover Operators	6	22	–	22
Population Factory	6	16	-	30
MOEA settings	14	–	9	–
Decoding solution	12	–	–	–
Total	80	109	9	149

Table 1: Implementation complexity for jMetal / Polymer. M.Loc=MOEA Expert LoC / D.Loc=Domain Expert LoC

domain itself. To evaluate the complexity of our approach compared to classical MOEA development we implement the cloud case study in both approaches and count the required lines of code (LoC) associated for each expertise. We keep the same coding style for both implementations in order to keep the absolute number of LoC comparable, however the most important point is the balance between MOEA and domain expertise. The Polymer implementation takes 158 LoC, while the implementation with jMetal needs 189, as shown in table 1. This difference is mainly due to the encoding/decoding in the jMetal implementation. Additionally, it is important to notice that the 24 LoC for the domain definition of the Polymer version are reusable for other model-based developments. For jMetal, in every operator, the genetic encoding must be defined by MOEA experts, in order to be usable by domain experts. That explains the 71 LoC difference for the MOEA experts. For Polymer, MOEA experts have just to decide the core settings (the algorithm and its configuration, number of generations, number of individuals in the population). However, the 40 additional lines difference for the domain experts in the Polymer implementation are due to the model structure and manipulations of objects where in the traditional implementation the structure is fixed in the array encoding. We can conclude from this subsection that the Polymer based implementation successes to reduce the development complexity by allowing developers to focus on cloud optimization (the actual domain problem).

5.2 Evolutive Refactoring Robustness

In order to evaluate the necessary refactoring effort for both platforms, we consider 10 different modifications (in 3 categories) on the previously defined case study. We then count the modified lines of code to adapt the implementation for both frameworks. Additionally, we check if the necessary changes can be pointed out by the Java type checker or not.

In the first category, we remain in the same optimization problem definition, but consider adding/removing fitness functions, mutation, and crossover operators, over the same problem. Both, Polymer and jMetal are able to check the type of the operators.

In the second category, we remain again in the same optimization use case but we change some parameters, *e.g.* adding/removing an application. In jMetal it is necessary to manually add/remove this application from the array of applications in each crossover/mutation operator and fitness function. In Polymer, we just need to add/remove this application from the population creation factory. We do not need to modify any operator or fitness because the model is integrally passed to them. Another modification that falls into this category, is to add an optimization field. Let's say we want to optimize over CPU and network resources of the deployed applications on the virtual machines. In Polymer, we just have to add a field "network" in the metaclasses $Task$, $Application$ and $VmInstance$. There is no code change in any previously implemented fitness function, mutation and crossover operator necessary. However, in the traditional approach a big change needs to be **manually** done on the encoding. Figure 2 is not enough anymore to represent the new encoding. Instead, a new integer needs to be reserved for each application task on each VM instance. The total number of integers in the array representing a solution, changes from the previously calculated $m \times (n+1)$ to the new value of $m \times (2n+1)$. Developers then need to manually update **all** previously defined operators in order to take this structural change in the encoding into account. In this simple use case implementation, we already counted as many as **43** lines of code affected by this change. What is even more dangerous, is that it is up to the encoding designer to define what comes first in the encoded array, among the v-cpu weight and the network weight. Type checkers cannot enforce that the encoding is actually used by its intended meaning in fitness functions and mutation operators. Table 2 summarizes the modifications.

5.3 Performance and Effectiveness

To evaluate the effectiveness of our approach we run both implementations (model-based and traditional MOEAs) 100 times on our cloud case study on a core i7 computer (2.7Ghz) with 2 GB RAM dedicated for the optimization process. As shown in figure 6 both

Modification	jMetal		Polymer	
	LoC	T. Check	LoC	T. Check
Adding Mutator	3	✓	1	✓
Removing Mutator	3	✓	1	✓
Adding Crossover	3	✓	1	✓
Removing Crossover	3	✓	1	✓
Adding Fitness	1	✓	1	✓
Removing Fitness	1	✓	1	✓
Adding an App	10	✗	1	✓
Removing an App	10	✗	1	✓
Adding Network Optim.	43	✗	1	✓
Changing Optim. Problem	All	-	40%	-

Table 2: Lines of code changed in both frameworks

approaches lead to similar Pareto fronts. We validate this using a Mann-Whitney U-Test (statistical test at 0.01 significance level) which give evidence that both Pareto are comparable. At the core both frameworks use the same MOEA algorithm, namely NSGA-II. Therefore, we can conclude that the bias introduced by the modeling layer does not impact the result.

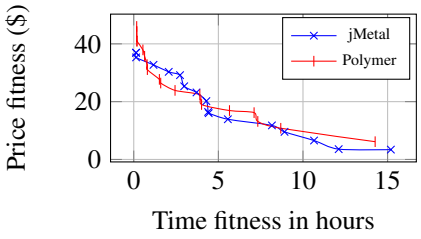

Figure 6: Pareto fronts for jMetal and Polymer

In order to evaluate the efficiency of our approach we measure the time for performing the solution search and the used memory. All in all, Polymer takes more time (by an average of 15 %) and memory (by an average of 20 %). This is due to model manipulations and cloning of models instead of simply copying arrays. The overhead in performance and memory is a trade-off to achieve an easier and more maintainable MOO development. However, this overhead decreases in term of percentage, when the complexity or the number of fitness functions to evaluate increase. For each genetic generation step the model cloning operation has a fixed cost. In order to optimize this step we apply the partial model cloning strategy. When dealing with complex fitness evaluations, the cloning cost and the overhead in performance become negligible (less than 5 % for the partial clone, and less than 15 % for the full clone). Figure 7 shows how the performance overhead (using as reference zero the traditional ad-hoc encoding) changes when the fitness functions become more complex to evaluate. As can be seen in figure 7 the overhead introduced by the partial cloning and model operations, become more and more negligible with increasing complexity of the case study. This increase in complexity can be due to one of three factors: 1) Increase in the number of objectives to optimize, *e.g.* optimizing network, RAM, disk usage, latency, security, redundancy and price at the same time. 2) Increase in complexity of previously defined fitnesses, *e.g.* a more precise fitness function is implemented. 3) Increase in the complexity of the domain itself, *e.g.* adding more virtual machines or more applications to optimize leads to slower fitness evaluation per generation. We can conclude, that our approach can compete with traditional MOEA usage by using heuristics like partial cloning. We can also see that the overhead becomes negligible for complex optimization scenarios, which are the main target of our approach (because for them maintainability becomes critical).

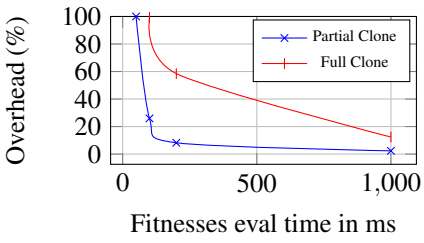

Figure 7: Performance overhead

6 RELATED WORK

Different encodings have been used in literature (Coello et al., 2002), *e.g.* binary, permutation, matrix, graph, and tree based encodings. They all require functions to encode, decode, navigate, and get information from the genetic encoded format. Many proposals have been presented for which encoding is suitable for which type of problem (Coello et al., 2002). For example, graph-based encodings are suitable for network optimization problems. Some encoding types, like permutation-based encodings, have the advantage to require only a very small memory footprint. None of these encodings are suitable for complex domains containing different types of data at the same time. Therefore, it is necessary to have a complex encoding/decoding step to transform the domain specific multi-objective optimization problem into a suitable MOEA representation and to transform the solution back to the domain problem. Recently,

ideas to apply MDE on MOEAs to simplify the encoding steps were presented (Williams and Poulding, 2011) and (Amato et al., 2014). Their idea is to create an automatic wrapper to transform a model into an array encoding and use the traditional approach behind. We discard the encoding step and use the model itself as encoding.

7 CONCLUSION

We claim that the usage of MOEAs to solve domain-specific MOO problems is complicated. This is mainly due to necessary encoding steps, meaning that a domain-specific problem must be mapped into a structure, which is suitable for the execution of MOEAs. This burdens developers with not only to understand a domain-specific MOO problem but also with the technical challenge to properly express this problem in terms of MOEA encoding. Many application domains can benefit from MOOs on top of models. In this paper we introduced a MDE framework to allow developers to combine MOOs with a model-driven development process. For this purpose, we enabled the execution of MOEAs on top of models and significantly simplified their usage and improved their reusability. We showed that the necessary MOEA expertise is hidden in the framework and enables software engineers to focus on domain-specific MOO problems instead of MOEA encoding and decoding. This has several advantages. First, MOO problems can be expressed in terms of domain models without being a MOEA expert. Second, type-safety is kept, which improves maintainability of the application. Third, since the problem is expressed in domain terms the solution is more readable. Our approach allows to use the same models for domain representation and MOO problem encoding to avoid the mismatch between these representations.

In future work we plan to integrate additional MOEAs in our framework, *e.g.* NSGA-III (Yuan et al., 2014). We also plan to apply the type information of models to introduce novel optimizations in MOEA algorithms (Elkateb et al.,). Finally, we will investigate the optimization of domain model cloning. As we have seen model cloning in our approach is still a bottleneck. A faster mechanism would improve the performance of our approach.

ACKNOWLEDGEMENTS

The research leading to this publication is supported by the FNR (grant 6816126), Creos Luxembourg S.A. and the CoPAInS project (code CO11/IS/1239572).

REFERENCES

Amato, A., Di Martino, B., and Venticinque, S. (2014). Multi-objective genetic algorithm for multi-cloud brokering. In *Euro-Par 2013: Parallel Processing Workshops*.

Auger, A., Bader, J., Brockhoff, D., and Zitzler, E. (2012). Hypervolume-based multiobjective optimization: Theoretical foundations and practical implications. *Theoretical Computer Science*, 425:75–103.

Coello, C. A. C., Van Veldhuizen, D. A., and Lamont, G. B. (2002). *Evolutionary algorithms for solving multi-objective problems*, volume 242. Springer.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE*.

Durillo, J. J. and Nebro, A. J. (2011). jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771.

Elkateb, D., Fouquet, F., Bourcier, J., and Le Traon, Y. Optimizing multi-objective evolutionary algorithms to enable quality-aware software provisioning. In *14th International Conference on Quality Software*.

Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., and Jézéquel, J.-M. (2012). An eclipse modelling framework alternative to meet the models@ runtime requirements. In *Model Driven Engineering Languages and Systems*, pages 87–101. Springer.

Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., and Jézéquel, J.-M. (2014). Kevoree modeling framework (kmf): Efficient modeling techniques for runtime use. *CoRR*, abs/1405.6817.

Frey, S., Fittkau, F., and Hasselbring, W. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *Software Engineering (ICSE), 2013 35th International Conference on*.

Kennedy, J., Eberhart, R., et al. (1995). Particle swarm optimization. 4(2):1942–1948.

Konak, A., Coit, D. W., and Smith, A. E. (2006). Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*.

Pandey, S., Wu, L., Guru, S. M., and Buyya, R. (2010). A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 400–407. IEEE.

Schaffer, J. D. (1985). Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st international Conference on Genetic Algorithms*, pages 93–100. L. Erlbaum Associates Inc.

Van Laarhoven, P. J. and Aarts, E. H. (1987). Simulated annealing.

Williams, J. R. and Poulding, S. (2011). Generating models using metaheuristic search. *Sponsoring Institutions*.

Yuan, Y., Xu, H., and Wang, B. (2014). An improved nsga-iii procedure for evolutionary many-objective optimization. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14.